4-2017

Characterization of Neural Network Backpropagation on Chiplet-based GPU Architectures

Colin A. Weinshenker
College of William and Mary

Characterization of Neural Network Backpropagation on Chiplet-based GPU Architectures

A thesis submitted in partial fulfillment of the requirement
for the degree of Bachelor of Science in Computer Science from
The College of William and Mary

by

Colin Weinshenker

Accepted for _____
(Honors, High Honors, Highest Honors)

_____
Adwait Jog, Director

_____
Zhenming Liu

_____
Dan Cristol

Williamsburg, VA
April 24, 2017

Abstract

Advances in parallel computing architectures (e.g., Graphics Processing Units (GPUs)) have had great success in helping meet the performance and energy-efficiency demands of many high-performance computing (HPC) applications. DRAM bandwidth is generally a critical performance bottleneck for many of such applications. With the advances in memory technology, the DRAM bandwidth bottleneck is shifting towards other parts of the system hierarchy (e.g., interconnects). We identify neural network backpropagation as one application where the interconnect network is one of the biggest performance bottlenecks. We show that the interconnect bottleneck for backpropagation can be significantly alleviated if computing cores and caching units are carefully tiled (an architecture commonly known as "chiplet") and organized on the interconnect fabric.

To simulate a chiplet design, we augment an existing, well-documented GPU simulator, GPGPU-Sim. Our modifications add an additional level of cache between on-chip L1s and an interconnect network-on-chip. This additional layer of cache reduces demand on the interconnect by localizing memory traffic to individual chiplets. We show that under a fixed core budget with additional cache, a chiplet architecture can increase Instruction Per Cycle (IPC) counts for important CUDA kernels by up to 20% during the training phase.

Acknowledgements

I would like to thank my advisor, Dr. Adwait Jog, for his encouragement and patience. His knowledge of the field shaped the project and taught me a great deal about having patience with myself. I appreciate Dr. Zhenming Liu's grounding feedback and the guidance of Dr. Dan Cristol, who has never failed to speak his mind. Lastly, thanks to Yorick Oden-Plants, Marissa Messner, and Lydia Boike, all of whom read drafts of this thesis. Yorick in particular helped keep me honest and optimistic.

Contents

1 Introduction

1.1 High Performance Computing

Over the past several decades, high performance computing (HPC) has become essential to nearly every field of scientific and commercial computing. Among forms of HPC, including cluster- and super-computing, none has risen to prominence faster than the graphics processing unit (GPU). The massive datasets that have emerged have proven well suited to the data parallelism GPUs readily exploit. From physics simulations to self-driving cars and modern gaming, GPUs are now nearly ubiquitous in modern computing.

In particular, GPUs have catalyzed the growth of commercial machine learning (ML). In the 1970s and 1980s, many academics and industry figures moved away from ML because the large models required for applications like speech and image recognition were too computationally expensive for available hardware. With the advent of commercially available GPUs, broad interest in machine learning awoke from hibernation during these "AI Winters" [10]. GPUs are particularly well-equipped to perform general matrix-matrix multiplication, the essential computation for machine learning. And GPU cost effectiveness (top-shelf GPUs sell for a few thousand dollars) has made them the tool of choice for deep learning, the branch of machine learning based around large neural networks.

In the past ten years, research on the algorithmic side of deep learning has exploded, producing better algorithms for neural networks and more computationally efficient ways of training them. Multibillion dollar companies like Baidu and DeepMind now work almost exclusively on ML research and development. Over the same period, research into the architectural design of GPUs laid the groundwork for a flourishing multiprocessor industry. GPU giants like NVIDIA, AMD, and Intel vie for market share with annual releases of new hardware with ever rising compute power and expanded feature sets. The result is that GPUs become cheaper and more ubiquitous with each new hardware generation.

Over all of this, developments in material science, electrical engineering, and chemistry look to disrupt the monolithic computer chip that has been the world's fundamental compute unit for nearly forty years. Traditionally, computer chips are produced on silicon disks called "wafers." Each wafer contains a "yield" of several functional, fully formed chips. Each chip is monolithic in that it is produced as several cores that cannot be separated without damaging the chip.

The new developments in relevant fields allow chip manufactures to cut a silicon wafer into several hundred self-contained compute components known as "chiplets." The circuitry of each chiplet is then "printed" individually [13]. This subdivision allows for "die-stacked"

architectures, heterogenous mixtures of individual cores and compute units. Chiplets constitute something of an economic revolution for the processor industry, as small- and mid-size businesses will at last be able to reenter a processor market closed to all but the largest of tech giants.

This project examines uses for chiplet architectures in the ML space. We show that backpropagation, a fundamental algorithm for training neural networks, can achieve performance on chiplet-based GPU architectures that outpaces backpropagation on traditional GPU architectures.

1.2 Contributions and Paper Overview

In this honors project, we make the following contributions:

- We identify the interconnect network-on-chip as a significant bottleneck to backpropagation in fully-connected neural networks.
- We outline the design modifications required to simulate chiplet-based architectures on an existing GPU simulator.
- We characterize the performance of backpropagation using core-isometric and core non-isometric variations on a simulated chiplet architecture.

We begin with a brief, informal discussion of some concerns about and uses for parallel programming. We then describe the basics of GPU architectures and the neural network algorithm (backpropagation) whose performance we characterize. We describe the design of our architectural solution and characterize backpropagation performance under the implemented solution.

2 Background

2.1 Parallel Programming in Brief

Before diving into GPUs, we must first have some understanding of parallel programming, what it offers, and what new problems and concerns it presents. Parallelism is in fact nearly ubiquitous in programming, and recognizing it is the first step toward making use of it.

With these ideas in mind, we consider a simple program.

$$a = b + c$$
$$d = e + f$$
$$g = a + d$$

Four independent integers (b, c, e, f) are summed in pairs. Then the sums are summed. So simple a program hardly seems worth investigation, but the parallel programmer will notice that this program can be executed in two steps rather than three. We cannot compute g until we have computed a and d. But the operands of a and the operands of d are independent. Provided we have the processing resources, there is no *a priori* reason not to compute a and d simultaneously. Our new two-step version of the program looks like this:

$$\left.\begin{aligned} a &= b + c \\ d &= e + f \end{aligned}\right\}$$
$$g = a + d$$

Serial programmers are accustomed to concerns of time complexity, the quantification of algorithmic runtime as a function of input size. Algorithms with poor runtime complexity are generally considered bad practice because they run for too long given large inputs (defining "too long" is generally a practical matter). Parallel programming introduces programmers to two new concerns: step complexity and work complexity. Step complexity is the quantification of steps required to complete an algorithm as a function of input size. Informally, we may consider step complexity the parallel world's analog to time complexity. So long as we are forced to perform operations one by one, regardless of data (in-)dependence, time is a natural measure of algorithmic complexity. But in the world of parallel algorithms, any number of independent operations may be performed at a time, so a single operation is an inadequate stand-in for time. In the parallel world, we measure time with steps (dependent operations) instead.

As an example of step complexity, we consider a variation on our sample program. This time we compute g as the sum of four intermediary sums: a and d as before, but also h, defined as the sum of arbitrary integers k and l, and j, the sum of arbitrary integers m and n. Again, our intermediary sums are computed in parallel.

$$
\left.
\begin{array}{l}
a = b + c \\
d = e + f \\
h = k + l \\
j = m + n
\end{array}
\right\}
$$
$$
g = a + d + h + j
$$

The new version achieves more than the original, but in two steps as before. No matter how many independent sums we compute in step 1, the program always terminates in two steps. We thus say that program step complexity is constant.

Work complexity refers to the amount of computation an algorithm requires as a function of input size. For an example, we consider a close relative of our original program. Again, we want to compute a sum reduction of some input integers. This time, though, our input is an arbitrary length array of random integers.

Algorithm 1: Serial Array Sum Reduction

Data: input: an array of random integers, n: the length of the input array

Result: A sum reduction of input

$sum = 0$;

foreach $i \in [0, 1, \ldots, n-1]$ **do**

  $sum \mathrel{+}= input[i]$;

return sum;

We must consider every one of the input integers in order to compute the sum, so the time complexity of serial sum reduction algorithm is $O(n)$. But we can use a parallel algorithm to perform sum reduce with step complexity of $O(\log_2 n)$.

Algorithm 2: Work-Efficient Parallel Sum Reduce

Data: input: an array of n random integers, n: the length of the input

Result: A sum reduction of input

foreach $i \in [\frac{n}{2}, \frac{n}{4}, \ldots, 1]$ **do**

  foreach $j \in [0, 1, \ldots, i-1]$ **do in parallel**

    $idx = i + j$;

    if $idx < n - 1$ **then**

      $input[j]$ += $input[idx]$;

return $input[0]$;

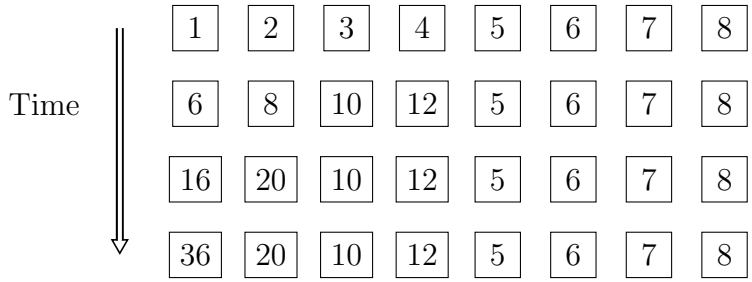


Figure 1: Work-efficient parallel sum reduce over integers 1-8

Parallel sum reduce has improved step complexity over serial sum reduce ($O(\log_2 n)$ vs. $O(n)$). This was achieved without increasing the number of additions required or adding redundant computation. We thus say the the algorithm is *work-efficient*.

Consider a variant of parallel sum reduce.

Algorithm 3: Work-Inefficient Parallel Sum Reduce

Data: input: an array of n random integers, n: the length of the input

Result: A sum reduction of input

foreach $i \in [1, 2, 4, \ldots, \log_2 n]$ **do**

  foreach $j \in [0, 1, \ldots, n-1]$ **do in parallel**

    $idx = i + j$;

    if $idx < n - 1$ **then**

      $input[j]$ += $input[idx]$;

return $input[0]$;

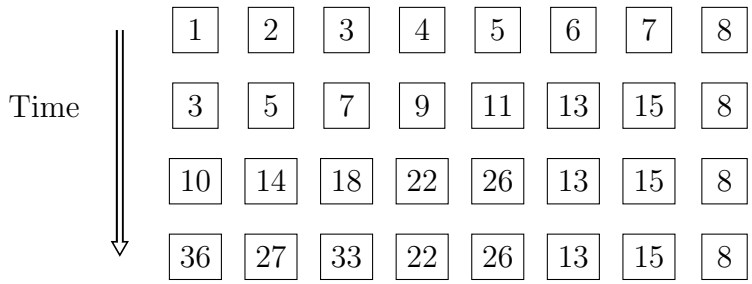


Figure 2: Work-inefficient parallel sum reduce over integers 1-8

The work-inefficient version achieves the same results as the first parallel sum reduce, but it performs superfluous additions on array elements five through seven. We thus say that this version is *work-inefficient*.

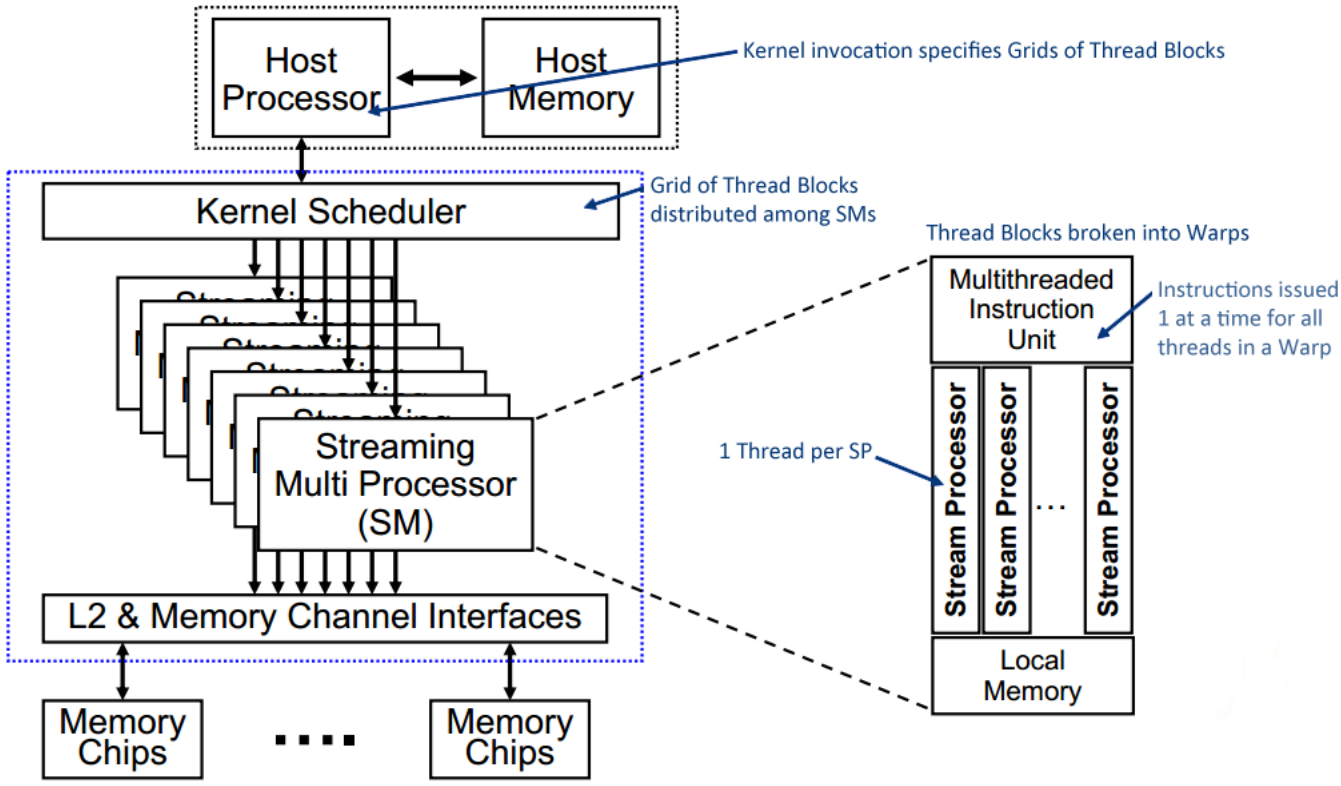
With some understanding of the concerns of parallel programming, we are prepared to address the uses of GPUs and their technical details.

2.2 The Graphics Processing Unit (GPU)

GPUs are massively parallel processors. Whereas the CPU found in the average laptop consists of several (usually one to four) powerful cores working together at high clock speeds, GPUs have dozens to thousands of cores executing at much slower clock speeds. This makes the GPU a poor choice for workloads with lots of serial data dependence but ideal for workloads that demand thousands or millions of data-independent computations. In Section 2.1 we saw that a parallel algorithm can improve theoretical bounds on runtime. In the case of parallel sum reduction, for example, using GPUs can have huge performance benefits for large input sizes.

2.2.1 SIMT Model

Figure 3: High Level View of a SIMT Model GPU [16]



For this project, we use a simulated NVIDIA Fermi architecture GPU. NVIDIA's GPU architecture is based on the SIMT (Single Instruction Multiple Thread) model. In the SIMT model, GPU cores dispatch units of work in groups called "warps" (named after loom warps, the original data parallelism). Each warp contains many "threads," with each thread responsible for a single parallel execution context of a program. As an example, consider a program that multiplies two 5-by-5 matrices. To accomplish this task, we may launch a single warp of twenty-five threads and make each thread responsible for generating one element of the output matrix. Each thread computes the dot product of its corresponding row and column in the input matrices and stores the result in the correct location in the output matrix. All twenty-five threads finish at nearly the same time, and the matrix multiplication is complete. A CPU performing the same matrix multiplication has essentially only one thread (albeit a very fast one). And while we may expect that a CPU may achieve comparable performance when multiplying two 5x5 matrices, as the size of the matrices increases a GPU's runtime will soon outperform a CPU's.

Within a GPU, thread warps are issued from SIMT cores, which are arranged in clusters of

one to several. As shown in Fig. 3, each SIMT core contains several streaming mutiprocessors (SMs). Each SM has multiple stream processors and access to per-thread memory. SMs delegate threads to stream processors, each of which manages a single thread and executes that thread's instructions in parallel.

SMs handle the problems of race conditions by executing instructions in lockstep. That is, all threads in a warp finish executing the current instruction before the next one is issued. Thread execution is masked to account for branching code. For example, a warp of thirty-two threads may assign each thread a random number, then instruct all threads whose number is greater than ten to double their numbers. During the doubling instructions, threads whose numbers are less than ten sit idle until the doubling instructions have finished for all threads. Then all threads in the warp resume lockstep execution.

2.2.2 GPU Memory Hierarchy

Each SM in a GPU has an attached first-level (L1) cache, as well as thread-specific memory and memory shared across threads in an SM. Though essential to CPU performance, lower levels of cache memory are not as common among GPUs. Furthermore, it is far more common to find two levels of cache among GPUs than the three levels that are par for modern CPUs.

GPUs also sport high bandwidth memory technology. In the first white paper for the release of Fermi architecture GPUs, it was noted that the Intel Nehalem architecture CPU, then the most advanced x86 CPU architecture, was capable of 32GB/s, a "commendable figure for a PC processor" [5]. DRAM bandwidth for an NVIDIA Fermi Architecture GPU is 144GB/s.

2.2.3 GPUs from the Programmer's Perspective

Current GPU hardware places the burden of identifying available parallelism on the programmer. Once a programmer identifies a code segment that would benefit from parallelization, she refactors her code into GPU-usable code segments (known as kernels) using architecture-provided extensions to common programming languages. On NVIDIA GPUs, these extensions and their interfaces to NVIDIA hardware are called the Compute Unified Device Architecture – CUDA for short.

Figure 4: CUDA Programming Flow [3]

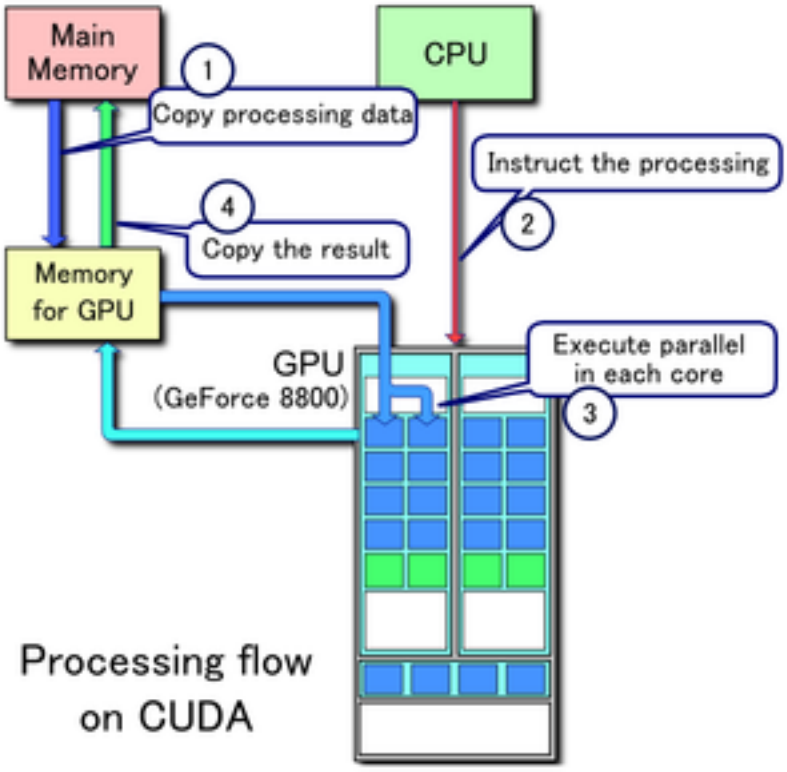
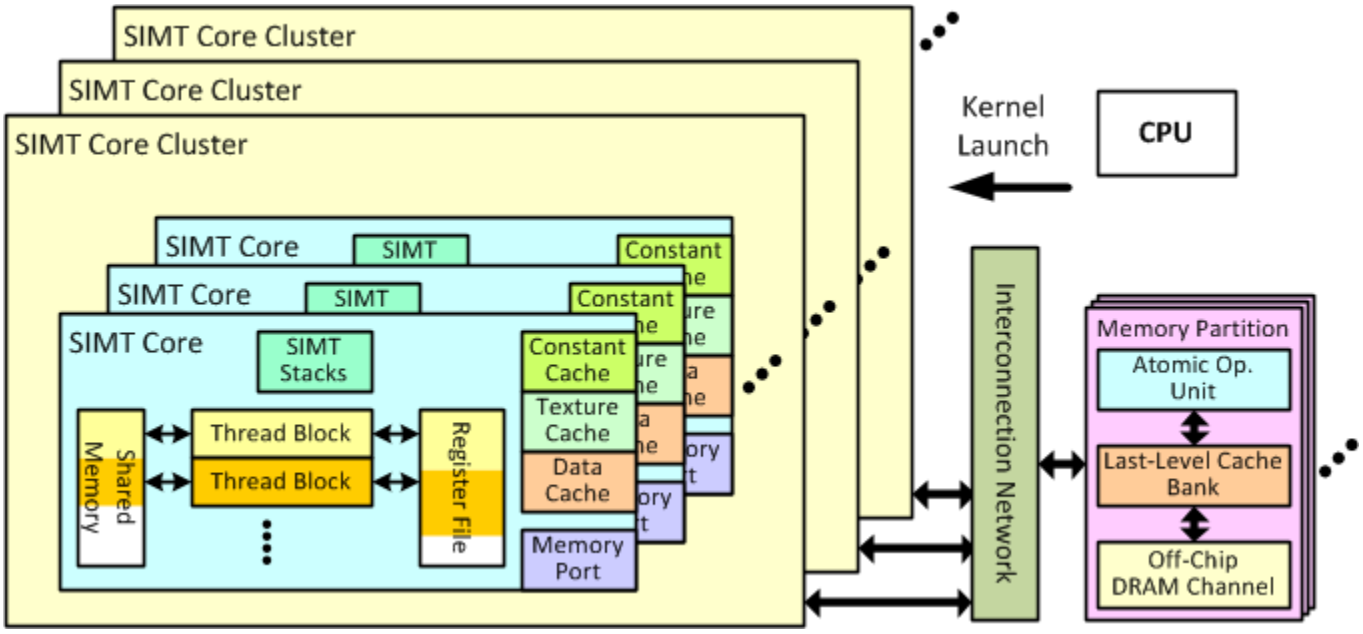


Fig. 4 depicts the flow of using CUDA-based GPUs. At program runtime, a programmer interacts with the GPU by first copying requisite memory (e.g., a pair of matrices) from the "host" CPU onto the "device" GPU. She then executes her kernels on the GPU, copies output from the device back to the host, and frees device memory.

2.2.4 GPU Simulator: GPGPU-Sim

We use an established GPU simulator, GPGPU-Sim [4], as the starting point for characterizing GPU applications. GPGPU-Sim can simulate several different CUDA architectures but unfortunately none from NVIDIA's recent Maxwell or Pascal hardware generations. We simulate a Fermi architecture GPU for its relative recency and architectural similarity to GPUs now on the market.

Figure 5: Overview of GPGPU-Sim architecture [18]



GPGPU-Sim concurrently simulates three different systems: 1) CUDA-Sim, a behavioral simulator of the CUDA PTX instruction set architecture, 2) GPGPU-Sim, a GPU architecture simulator, and 3) Intersim, an interconnect network simulator adapted from Jiang et. al's BookSim [8].

Combined, these systems form a cycle-by-cycle simulator of GPU hardware capable of running most CUDA 4 applications. GPGPU-Sim is extremely detailed and logs information on hardware performance from per-kernel IPC counts to full cache statistic reports. GPGPU-Sim is also built for extensibility. Implementing hardware modifications or additional statistics to track is relatively straightforward.

However, GPGPU-Sim suffers from long simulation times. Running realistically sized GPU workloads on a CPU, with added overhead for data collection, is simply too much for most applications, and this affected our study of backpropagation (see Section 3.2).

2.3 Interconnect Networks

Interconnect networks are communication media that make parallel processing possible. With traditional CPU bus interconnects, only one processor can access the interconnect at a time. Such restricted access to memory is anathema to parallel architectures' goal of latency masking, so more sophisticated interconnect designs are employed to allow parallel access to memory. In the context of GPUs, the design of an interconnect network determines

how SIMT core clusters and memory controllers are spatially arranged and the efficiency with which they communicate. In measuring that efficiency, we consider both latency and throughput.

An interconnect network is a graph consisting of routers and links (vertices and edges) connecting network endpoints, or "nodes." Data moves along the network via transmission points ("routers") connected by signal-bearing wires ("links") in units of transfer known as "packets." Each packet consists of one or more flits, an atomic unit of data flow. In GPGPU's implementation of interconnects, each router performs a multiplex operation to determine which of its links, if any, transmits a flit during every interconnect cycle.

Integral to network design is network *topology*. Interconnect topology is a rich subject combining network theory and information theory, and it determines many important theoretical and practical features of interconnects. For example, topology partially determines the hop count (the number of routers a packet must travel) between any two nodes. For our part, we consider only two commonly used topologies: 2-D mesh and crossbar.

Figure 6: 6x6 Mesh Interconnect for GPGPU-Sim [2]

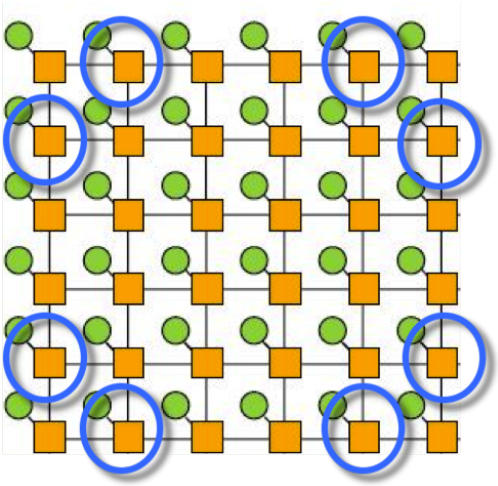
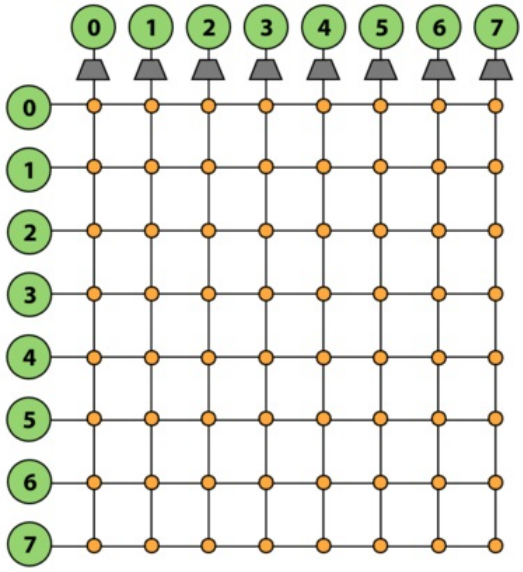


Fig. 6 represents the layout of the 6x6 2-D mesh we use to simulate neural network backpropagation. Each orange square represents a router on the network, and each router has two, three, or four outgoing links (black lines). The green circle connected to each router not inside a blue circle represents a SIMT core cluster, and the blue-circled routers represent the locations of memory controllers holding last-level caches and connections to DRAM.

Crossbar topology is even simpler than mesh. A crossbar fully connects n inputs to m outputs, so the hop count between any two points on the network is 1. Crossbars are

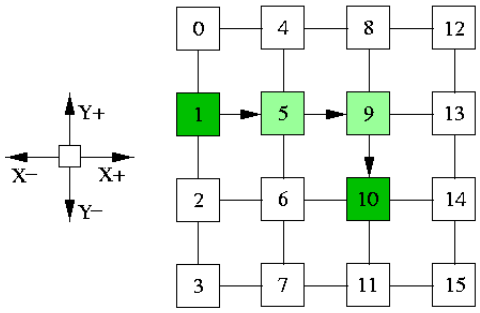
typically low-latency, high-bandwidth topologies, but they require mn routers and are poor choices for large-scale architectures.

Figure 7: Crossbar Interconnect [2]



Flits are routed across the network according to a policy known as the routing algorithm. A good routing algorithm attempts to balance the load faced by the network over time to prevent performance bottlenecks or application deadlock. Routing algorithms come in both deterministic and non-deterministic flavors, and the optimal algorithm for a given use case varies with application memory access patterns and interconnect topology. For our part, we consider only two routing algorithms, one for each interconnect type. For crossbar networks, we use destination tag routing, which determines the target endpoint by a mask on the packet header. For mesh networks we use XY routing, a deterministic, dimension-order routing algorithm. Under XY routing, flits arrive at their destination by traveling first along the X dimension, then the Y dimension. Fig. 8 shows an example of a flit traveling between two nodes under XY routing.

Figure 8: XY Routing [19]



3 Machine Learning

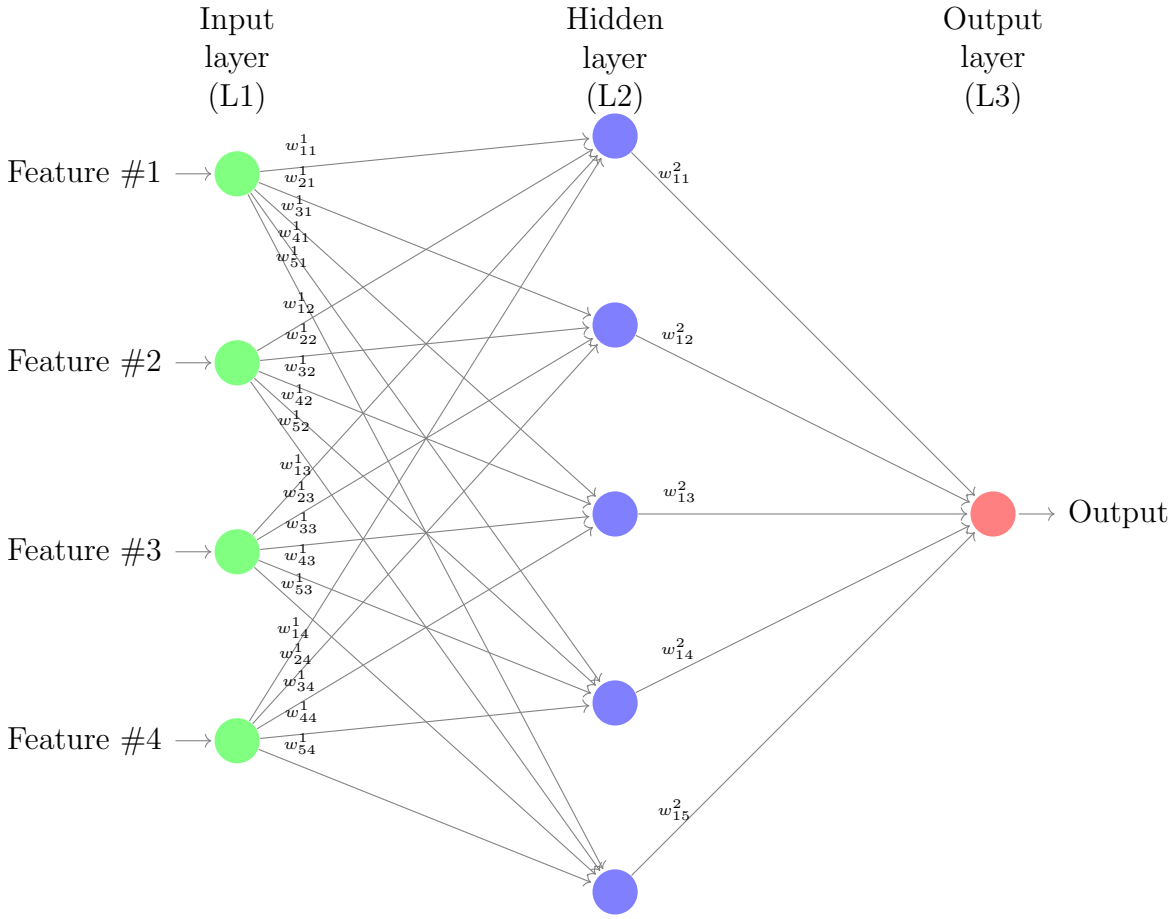
Machine learning refers to a broad range of computer science ideas and techniques that enable computers to "learn" data models without explicit programming. In today's ML landscape, none of these ideas and techniques garners more attention than neural networks.

3.1 Backpropagation

Artificial neural networks are a broad category of machine learning algorithms, all of which are in some aspect designed to mimic the activity of the brain. Real neural networks consist of neurons and their connections. Similarly, artificial neurons consist of neurons (really non-linear functions) with weighted connections (represented by matrices). Backpropagation (backprop, for short) refers to a set of algorithms for training artificial neural networks to fit data models. Which form of backpropagation is used varies by the species of neural net in use, but in all its forms backprop breaks down into two phases: feed-forward and update.

In the feed-forward phase, observations in the form of n-dimensional feature vectors are passed to the input layer. Each layer multiplies its inputs by the weight matrix connecting it to the next layer. A non-linear transformation (e.g., a sigmoid or rectified linear unit function) is applied to the result, and the process is repeated, propagating the input signal through the network. The transformed signal that reaches the output layer is interpreted as an encoded output – an image classification, for example.

Figure 9: A Multi-layer Perceptron (MLP) network. w_{ij}^{k} denotes the weight from neuron j in layer k to neuron i in layer $k+1$



In the weight update phase, the correct output for the given input is used with a loss function to compute the error of the network's prediction. A simple squared error function can suffice.

$$E = \frac{1}{2}\sum_{i=0}^{n}(t_i - y_i)^2$$

Where E is the sum of the network's prediction error over all n training examples in the training set, t_i is the true label for input sample i, and y_i is the network's predicted classification for input i.

After determining the prediction error on an observation, the weights of the network are updated. Since the functions by which the inputs determine the error of the network and their gradients with respect to the network's last predicted output are known, the chain rule

can be applied to each function in the network to create a map of how network error changes with respect to any individual weight [9] [14]. Network weights are then updated to improve network performance for the last seen observation.

To train a neural network practically, one feeds it a labeled "training" dataset (i.e., a dataset consisting of input features and known correct outputs). For each input, the error of the network's output is computed. After summing the error over a certain number of observations (referred to as the mini-batch size), the weights are updated. Over many updates, the weights in a network form a logical structure relating inputs to probabilities for expected outputs.

Backprop is typically performed in conjunction with an optimization method such as gradient descent so the network satisfies a local optimum for accuracy given the network structure, loss function, and training set. A network whose weights have converged on a local minimum of the loss function is not necessarily field-ready, however, and the design of neural networks with robustly low error on training, validation, and test sets is the subject of open research and, to some extent, artistry.

3.2 Characterization of Neural Network Backpropagation on GPU Architectures

We use a GPU-optimized implementation of backpropagation on fully-connected neural networks from the GPUMLib suite [12]. This implementation is broken down into six kernels: Fire Layer Neurons (FLN), Fire Output Layer (FOL), Root Mean Square Error (RMS), Robust Learning (RL), Calculate Local Gradients (CLG), and Correct Layer Weights (CLW).

As mentioned in Section 2.2.4, simulation times for GPGPU-Sim applications are a serious hindrance to our methods of studying GPU applications. We thus simulate neural networks that are relatively small compared to ones used for commercial applications (see Section 5.1). The average cycles GPGPU-Sim simulates during each kernel in a training pass, with and without last-level caches enabled, are shown in Fig. 10.

Figure 10: Cycles Simulated for each Kernel of GPUMLib Backpropagation

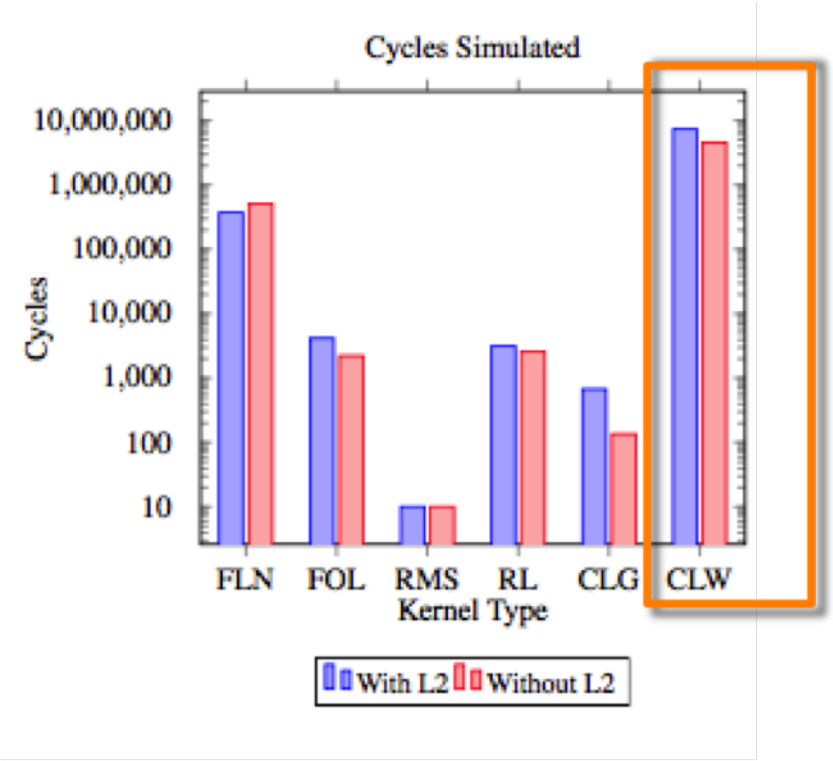


Table 1: CLW Performance Sensitivity to Minibatch Size

	Mini-batch size	
	32	256
IPC	248	101
Average Memory Fetch Latency	253	518
L1 Data Cache Miss Rate	0.17	0.47
Last-Level Cache Miss Rate	0.01	0.017
Average DRAM Bandwidth Utilization	0.01	0.02

In Fig. 10 we see that the update step (CLW) dominates training time. CLW averages roughly four times the cycles of next longest running kernel, FLN. Table 1 shows the sensitivity of CLW to increasing the mini-batch size. In GPUMLib's implementation of backprop,

mini-batch size directly corresponds to CUDA grid size (i.e., the number of sets of threads), so increasing the mini-batch is a way to see how backprop's memory problems are exacerbated by larger input sizes. We observe that kernel performance is low, and increasing the input size exacerbates the issues. Namely, IPC counts drop by 50% when increasing the mini-batch size to 8x. We also observe that the mini-batch increase has little impact on the lower level cache and DRAM performance.

Figure 11: Backpropagation Bottleneck Location in the GPGPU-Sim Architecture [18]

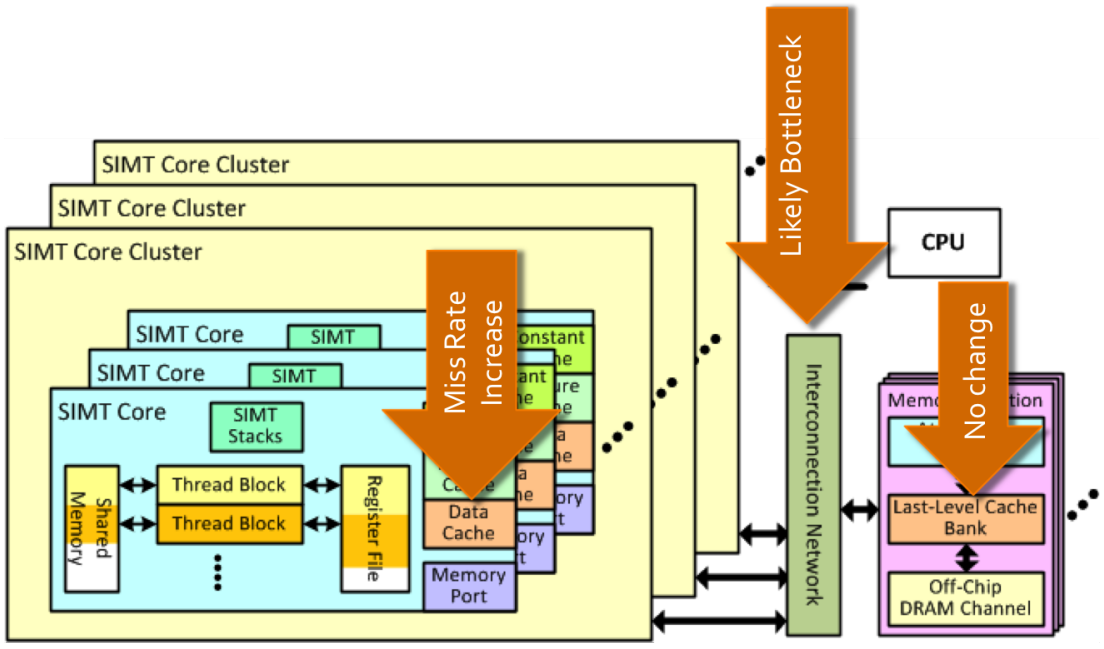
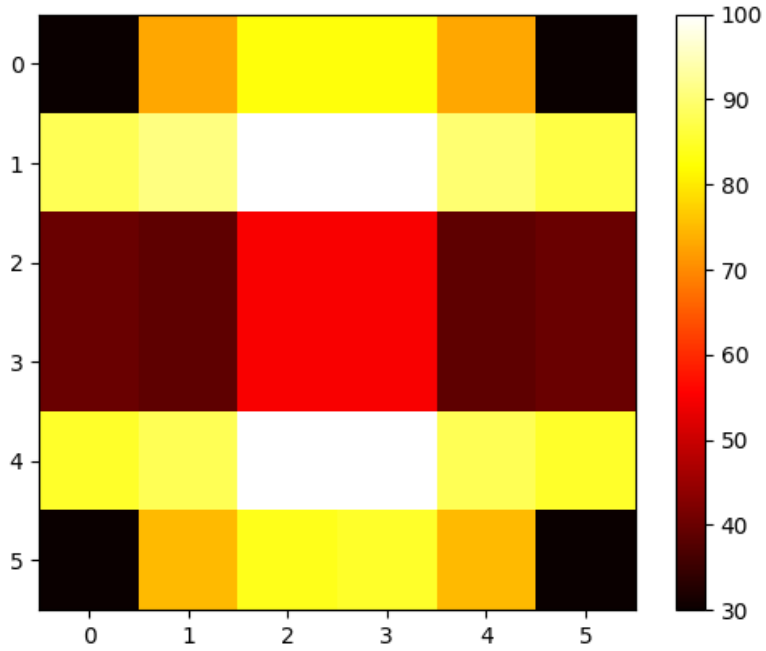


Fig. 11 shows the visual logic of our reasoning around backprop's poor performance with respect to GPGPU-Sim's simulated hardware. As IPC declines, the per-core data cache (left arrow) miss rate rises. But the last-level cache miss rate (right arrow) stays roughly constant. This suggests the presence of a memory bottleneck in the communication between caches. GPGPU-Sim uses Intersim to simulate an interconnect network-on-chip (center arrow) connecting per-core data caches and the last-level cache banks attached to memory controllers. We examine this interconnect as a possible application bottleneck.

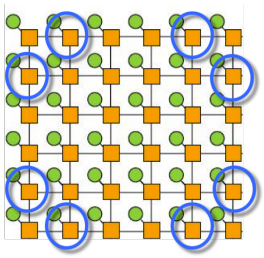
As was just mentioned, the cache behavior when running backprop on GPGPU-Sim indicates the presence of a bottleneck in the interconnect during the weight update kernel (CLW). To confirm this, we measure utilization for each of the 36 routers on the six-by-six mesh during CLW. Our per-router utilization metric divides the number of interconnect clock cycles in which any of the router's links are in use by the total number of interconnect

cycles. Fig. 12a shows the router utilization metric for the simulated mesh reply network during CLW. For reference, Fig. 12b repeats Fig. 6's arrangement of memory controllers (circled nodes) and SIMT core clusters (uncircled nodes) on the mesh network.

Figure 12: Reply Network Router Utilization For Weight Update Kernel [2]



(a) Router Activity During Correct Layer Weights

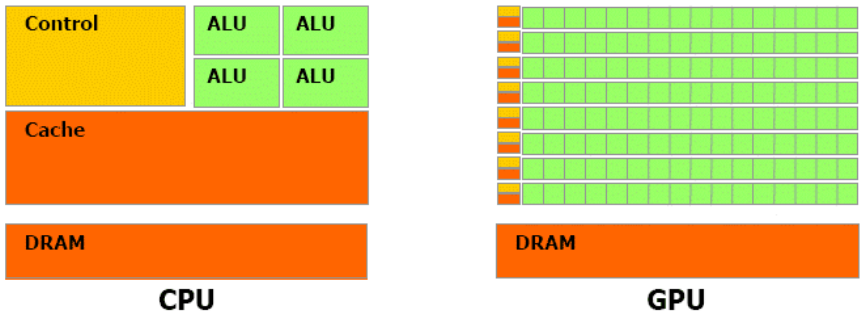


(b) Memory Controller Map

There are clear hotspots above and below the center four core clusters during CLW's interconnect clock cycles, indicating that network throughput is insufficient for the memory traffic during CLW. There is also heavy traffic through all non-corner routers surrounding the memory controllers. Taking the hotspots and Table 1's data into account, we conclude that the interconnect is a legitimate bottleneck. We attempt to alleviate this heavy interconnect traffic through implementation of a chiplet-based architecture.

4 Chiplet-based Architecures

Figure 13: Traditional Computer Architectures vs. Chiplet-based Architectures



(a) CPU Architecture vs. GPU Architecture [1]

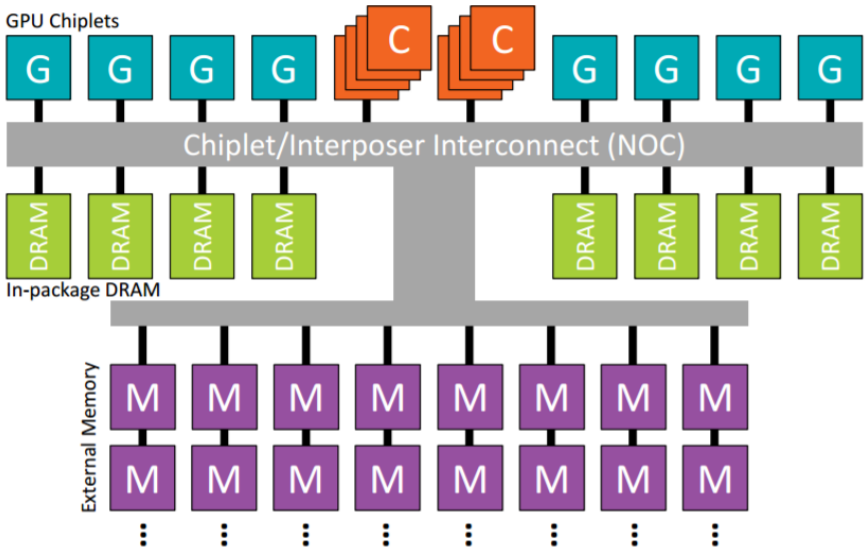


Figure 3. Block diagram of the ENA memory system

(b) Chiplet-based Architecture [20]

Fig.13a shows the differences between traditional CPU and GPU architectures. A CPU core sports a single control unit with several attached arithmetic logic units (ALUs), which perform arithmetic and bitwise operations. CPU cores have at least one (and typically three) levels of caches to exploit spatial and temporal data locality. For our purposes, GPU architecture can be thought of as CPU architecture scaled up. Each GPU core consists of many control units and caches (though typically only one level), with each control unit-cache pair connected to many ALUs (SMs). Both CPUs and GPUs are complete systems on chip (SOCs). That is to say that each is a fully functional compute unit, with all necessary

memory interfaces.

By contrast, chiplets are not in and of themselves complete SOCs. For example, a GPU chiplet may consist of several GPU cores and their associated caches but no connections to DRAM or external memory [20]. A chiplet-based architecture, shown in Fig.13b, takes many CPU and/or GPU chiplets and tiles them on an interconnect network along with connections to DRAM and external memory. Thus the arrangement of chiplets and DRAM on the interconnect, the design of the interconnect, and the structure of the chiplet memory hierarchy all become important to system performance.

A chiplet-based architecture has several benefits not found in architectures based on monolithic SOCs [20]. First, the methods for producing silicon dies have higher yield rates than traditional silicon wafers. Second, designing a monolithic SOC architecture to maintain high performance across a broad range of applications with different demands for task and data parallelism is exceedingly difficult and expensive; with chiplet architectures, individual components (e.g., CPU cores and GPU cores) can be optimized to the functions for which they are best suited and combined. Third, with proper design, heterogenous chiplet-based architectures can be decomposed into their constituent parts and repackaged for reuse.

4.1 Simulating a Chiplet-based Architecture in GPGPU-Sim

The switch from GPGPU-Sim as a simulator of traditional GPU architectures to chiplet-based architectures is largely a change of perspective. GPGPU-Sim already comes equipped with many features required to simulate a chiplet architecture, including a variable number of SIMT cores per cluster and an interconnect fabric with configurable tiling for memory controllers and SIMT clusters. In the switch from a simulator of a traditional GPU to a chiplet-based GPU architecture, we simply decide that the interconnect network now holds chiplets on its endpoints, rather than SIMT core clusters. (Note that our simulation of a chiplet-based architecture has GPU chiplets only.) We then consider what modifications to a standard SIMT core cluster might make it more effective as a chiplet. Interconnect congestion during backprop's update phase motivates localizing memory traffic to each chiplet. We thus add another layer of cache (L2) shared by all cores on each chiplet.

4.1.1 Cache Design

One major caveat is that difficulties with implementation led us to use a write-through cache as opposed to a likely more-efficient write-back implementation. For each chiplet, misses in the L2 are coalesced via miss status hit registers before being sent across the interconnect

to memory controllers. The L2 itself is also on the core clock cycle.

4.1.2 Queuing Mechanism

Ideally, we would have implemented a second interconnect network between the existing L1 caches and the shared L2 cache. Unfortunately, the Intersim package is not designed to simulate more than a single global interconnect, and modifying the software to allow for multiple interconnects was outside the scope of the project. In lieu of an added interconnect, we use two queues to move memory traffic between the L1 and L2 caches. Fig. 14 depicts this implementation. One queue passes memory requests from each GPU chiplet's L1 caches to its L2 cache. The other passes memory responses from the L2 cache back to the L1 caches. Another pair of queues handles communication between each chiplet's L2 cache and the interconnect network.

Since all cores in each cluster share one pair of queues, we were concerned that increasing the number of cores per cluster would degrade cache bandwidth between the L2 cache and the L1 caches, perhaps to the point where the queues themselves could become a bottleneck. Fortunately, this fear has not been borne out. There are likely still advantages to implementing a second interconnect, but the queue-based mechanism yields performance benefits for backprop provided there is sufficient budget for the L2 cache (see Section 5).

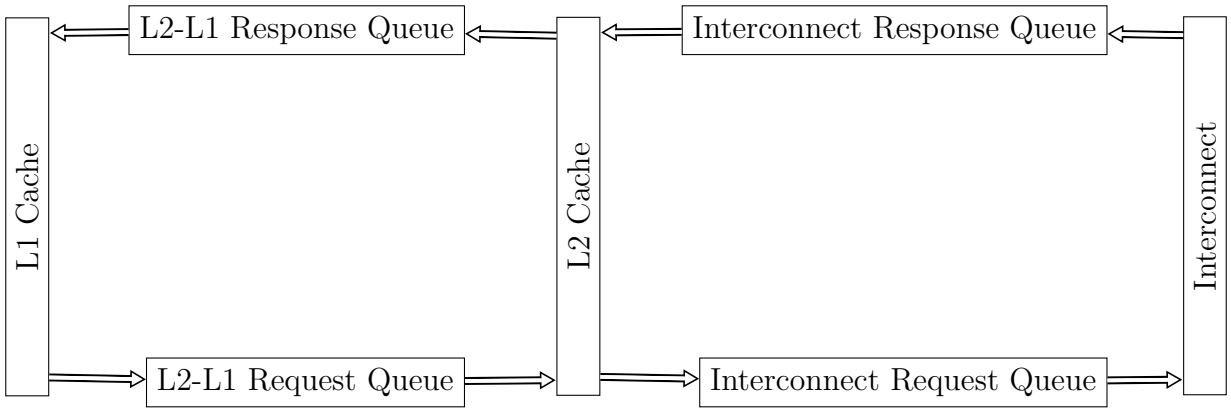


Figure 14: Implemented Chiplet Memory Hierarchy

5 Evaluation and Results

5.1 Methodology

We evaluate backpropagation performance on core-isometric and core non-isometric variations on a chiplet design. For all experiments we run backprop on a three-layer multilayer perceptron (MLP) network trained on the MNIST dataset [11]. The MLP layer dimensions are 784 (input) – 100 (hidden) – 10 (output), and we use a mini-batch size of 32 images. We then vary the compute power available (total number of SIMT cores), number of SIMT cores per cluster, and the sizes of L2 and L3 caches. For each training pass, we measure total IPC, memory fetch latency, L2 cache performance and L3 cache performance. In addition, we measure DRAM bandwidth utilization and interconnect utilization during just the weight update kernels. Finally, we consider the possibility that lower-level caches may be detrimental to GPU performance for neural network backprop by conducting trials with neither an L2 nor an L3 present.

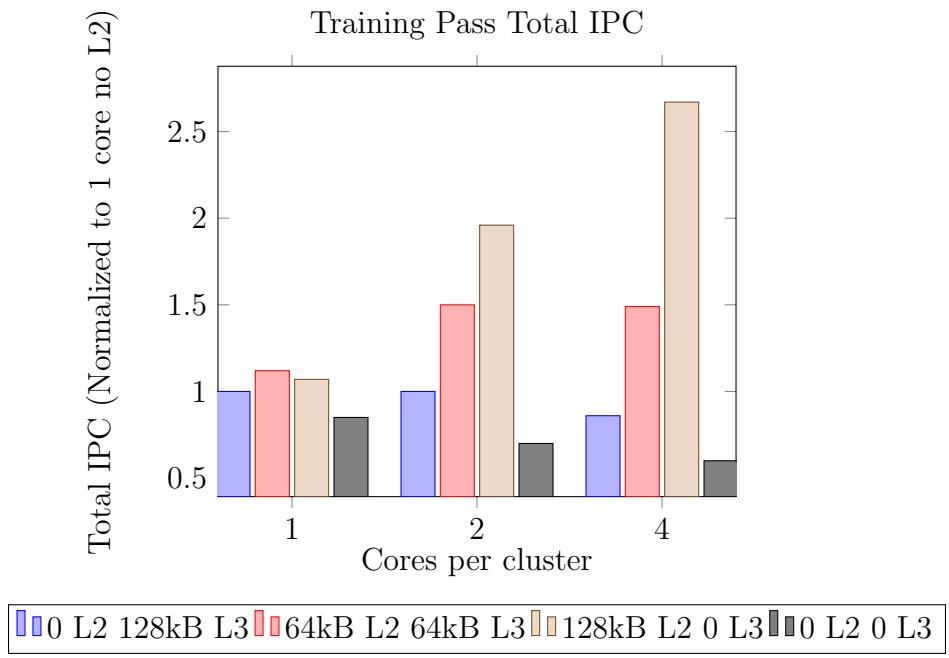
5.2 Results

5.2.1 Core Non-Isometric Results

By allowing the total SIMT core count to increase with the density of cores per cluster, we hope to show scalability in our design. When we increase the number of cores per chiplet on the mesh interconnect, we allow the total number of cores to increase. Recall that we use a 6x6 mesh with 8 memory controllers. So a 2-cores-per-chiplet-configuration, for example, has 56 total cores, while a 1-core-per-chiplet configuration has 28 cores. We use variations on L2 and L3 cache sizes up to 128kB each, but we do not fix the total available global cache size. That is, we use four configurations for the total size of available lower-level cache:

- 0 L2 and 128kB L3. Total lower-level cache $= 8 * 128kB = 1024$kB.
- 64kB L2 and 64kB L3. Total lower-level cache $= 36 * 64kB = 2304$kB.
- 128KB L2 and 0kB L3. Total lower-level cache $= 28 * 128kB = 3584$kB.
- 0 L2 and 0 L3. Total lower-level cache = 0 kB.

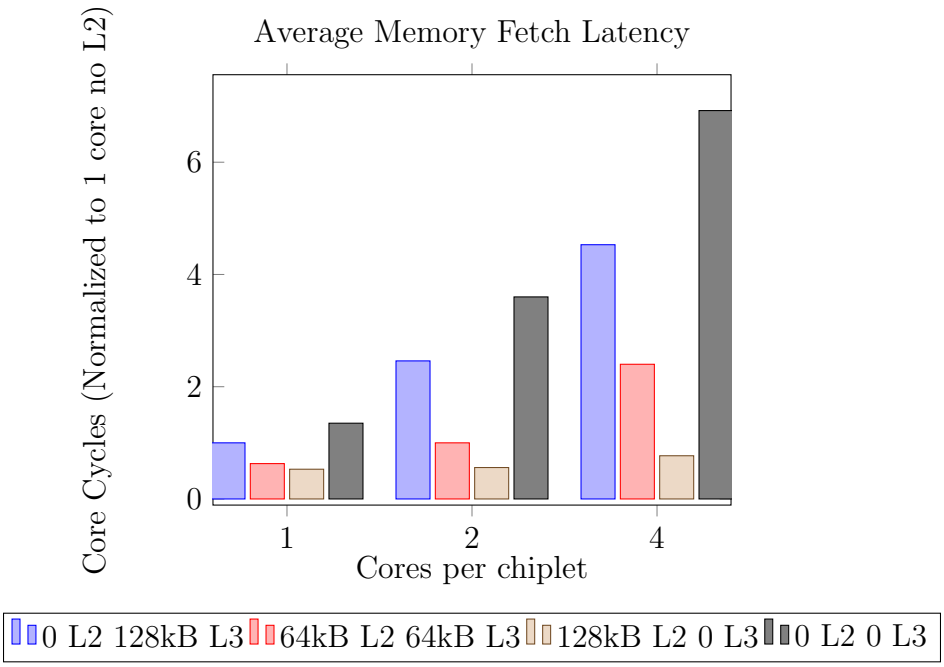
Figure 15: Backprop IPC Sensitivity to Core Density and Cache Distribution



To show that the chiplet architecture improves the utilization of GPU compute power, we measure the total IPC over one backpropagation training pass, normalized to the case of using a 128kB L3, no L2, and one core per chiplet. Fig. 15 shows the number of instructions per cycle executed (y axis) during the training pass varies with changes in the number of SIMT cores per chiplet (x axis) and distribution of cache (bar color). In Fig. 14 we see clear performance benefits to the added L2 cache, and these benefits increase with the density of SIMT cores per chiplet and the size of the L2. However, we observe that there is a point at which increasing the density of cores per cluster washes out the benefits of an amount of added cache. For instance, between densities of 2 and 4 SIMT cores per chiplet, IPC under the 64kB L2 64kB L3 configuration declines slightly.

Conversely, increasing the density of SIMT cores per chiplet is detrimental to GPU performance when no L2 is present. In both the 0 L2 128kB L3 configuration and the 0 L2 0 L3 configuration, IPC decreases even as the total available compute power (number of cores) increases.

Figure 16: Backprop Memory Fetch Latency to Core Density and Cache Distribution



To show the impact of the chiplet architecture on the latency of memory requests during backprop, we study how the latency of the average memory fetch (y axis) varies with the number of SIMT cores per chiplet (x axis) and the distribution of cache (bar color). We normalize all latencies to the case of 0 L2 128kB L3 with 1 core per chiplet. We expected that, as a general rule, memory fetch latency would increase with the density of SIMT cores per chiplet and decrease with the size of L2 cache across core-isometric configurations. As with Fig. 14, Fig. 15 shows the performance benefits of the added L2. With 128kB L2 caches and no L3 caches, backpropagation simulated with 4 cores per chiplet has lower memory fetch latency than backpropagation simulated with just one core per cluster and 128kB L3 caches and no L2 caches.

Figure 17: L2 and L3 Cache Miss Rate Sensitivity to Core Density and Cache Distribution

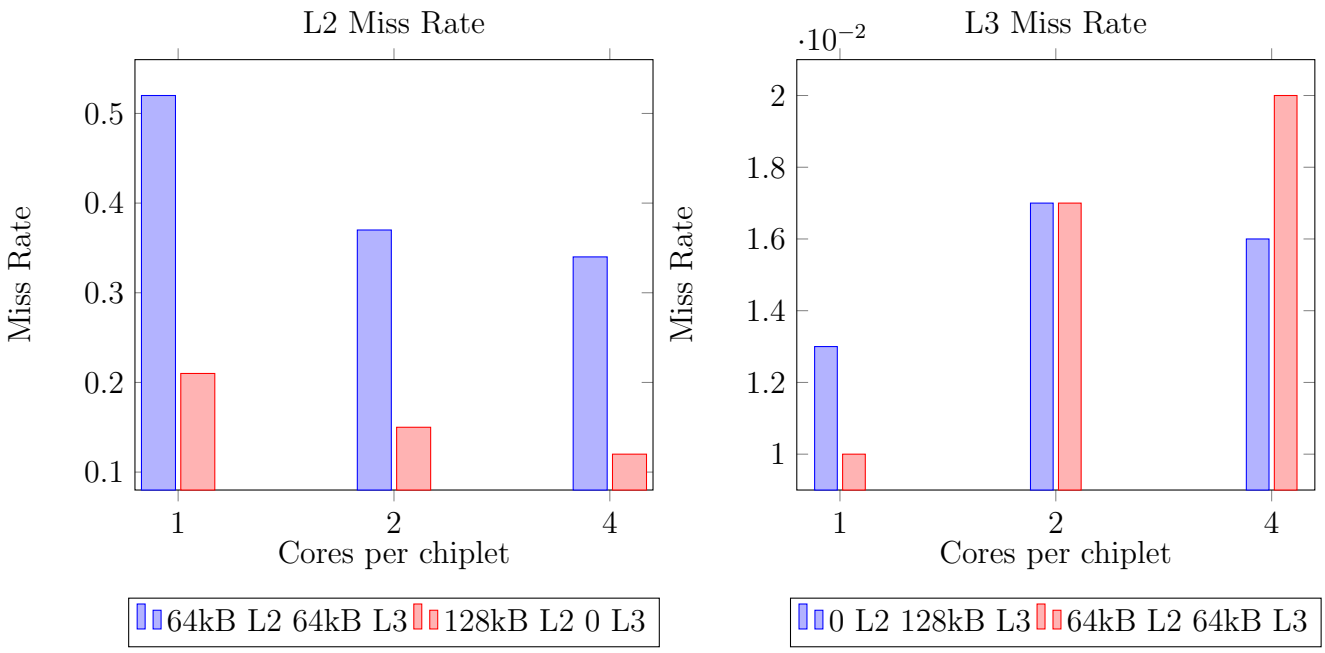


Figure 18: L2 Cache Access and Miss Sensitivity to Core Density and Cache Distribution

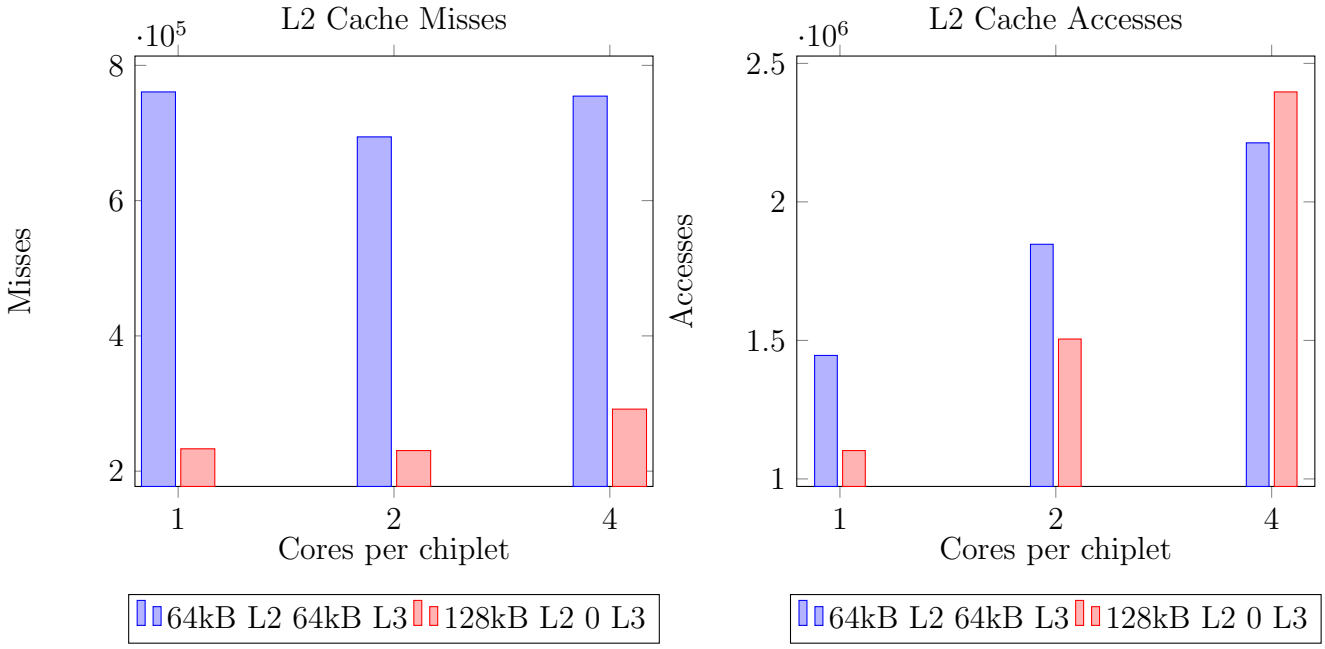
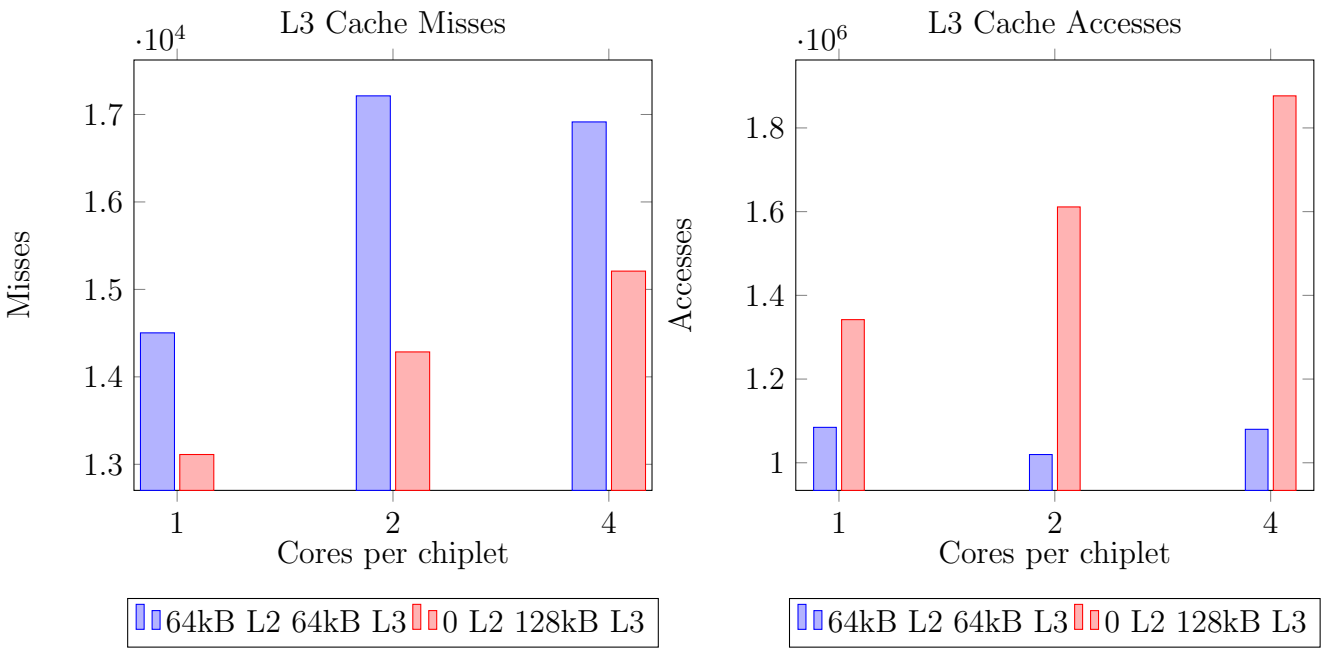
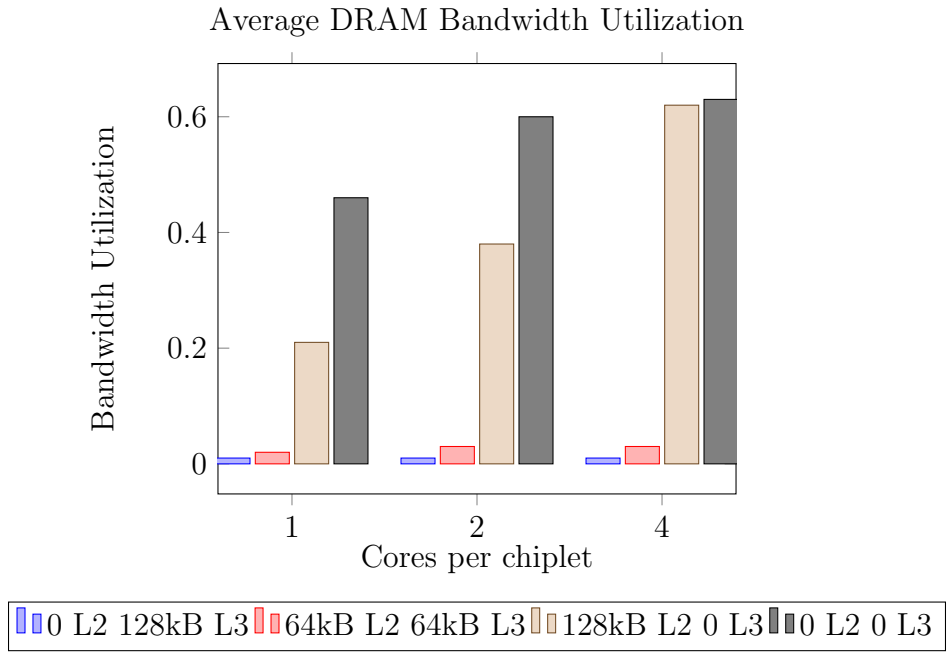


Figure 19: L3 Cache Access and Miss Sensitivity to Core Density and Cache Distribution



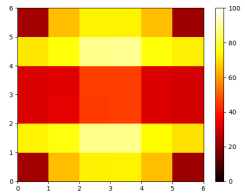
We study the L2 and L3 miss rates to ensure that varying the cache sizes (particularly the L3) does not change the performance of the lower memory hierarchy or create new bottlenecks. Cache miss and cache access statistics are collected to place the effects of the cache size in context of their respective magnitudes. Figs. 17, 18, and 19 show these statistics. Our goal in adding L2 caches is to significantly reduce the number of L3 accesses while maintaining low L2 miss rates. We observe that L3 miss rates are very low with or without L2 caches (see Table 1). In Fig. 17, the miss rate rate for L2 caches under the 64kB L2 64kB L3, 2 cores per chiplet configuration is under 40%, while the number of L3 cache accesses is nearly 60% of the number of L3 cache accesses when no L2 caches are present. As we noted with total IPC, there is a point at which the density of SIMT cores per chiplet overtaxes the available cache and performance no longer significantly improves. For example, between the 64kB L2 64kB L3 with 2 cores per chiplet configuration and the 64kB L2 64kB L3 with 4 cores per chiplet configuration, L2 miss rate decreases by just 3%, and the corresponding IPC improvement is marginal despite doubling the total number of available cores (Fig. 18, Fig.17).

Figure 20: CLW DRAM Utilization Sensitivity to Core Density and Cache Distribution

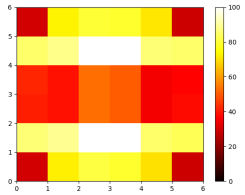


As we shrink memory controllers' L3 caches and grow chiplets' L2 caches, we do not want to reduce the total available L3 cache to the point that DRAM bandwidth becomes a bottleneck. We thus study the memory controllers' average DRAM bandwidth utilization during the weight update kernel as a function of cache distribution and cores per chiplet. Fig. 20 shows that DRAM utilization was not a problem under our configurations, except when L3 caches were removed entirely. Without L3 caches, DRAM bandwidth spikes significantly and increases with the density of cores per chiplet. Under our run parameters, however, 64kB of L3 cache per memory controller is sufficient to hold DRAM bandwidth utilization at just over 2%, even with 4 cores per chiplet.

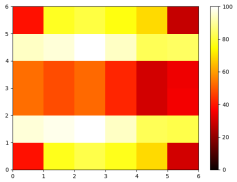
Figure 21: Mesh Interconnect Reply Network Router Utilization During Weight Update



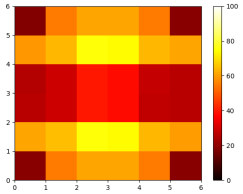
(a) 128 kB L3 1 core/ chiplet



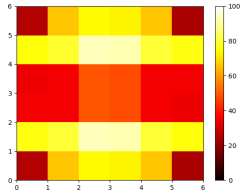
(b) 128 kB L3 2 cores/chiplet



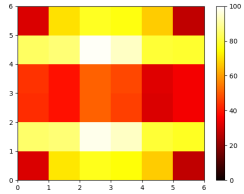
(c) 128kB L3 4 cores/chiplet



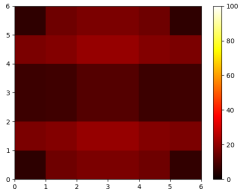
(d) 64kB L2 1 core/chiplet



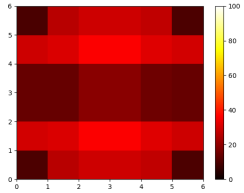
(e) 64kB L2 2 cores/chiplet



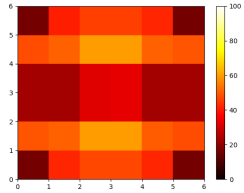
(f) 64kB L2 4 cores/chiplet



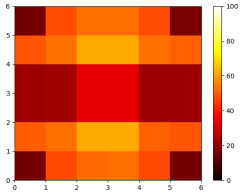
(g) 128kB L2 1 core/chiplet



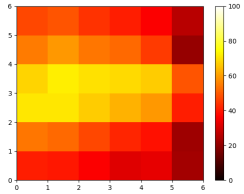
(h) 128kB L2 2 cores/chiplet



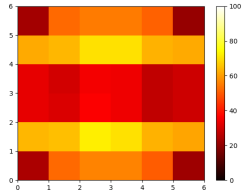
(i) 128kB L2 4 cores/chiplet



(j) No last level cache 1 core/chiplet



(k) No last level cache 2 cores/chiplet



(l) No last level cache 4 cores/chiplet

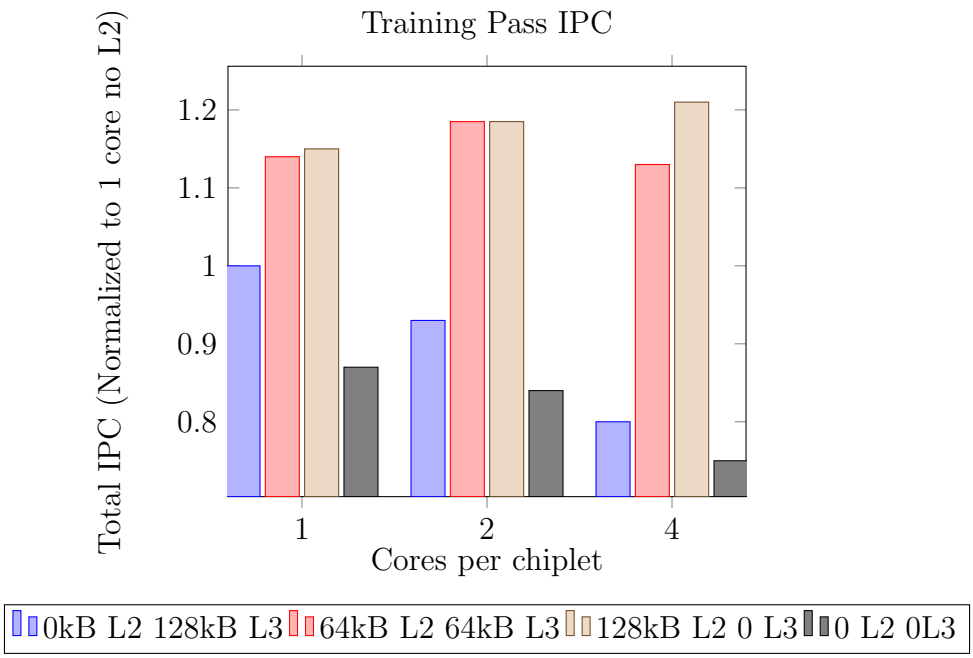
Since alleviating the interconnect memory bottleneck was our original motivation for adding L2 cache, we measure router utilization for each of our configurations using the heat map format from Section 3.2. Each row of heat maps shows the effect on the mesh interconnect utilization as the number of cores per chiplet increases between cache-isometric configurations. Each column shows the change in interconnect utilization as the cache distribution changes between core-isometric configurations. We observe that, across all configurations, increasing the size of L2 caches alleviates the interconnect bottleneck. The heat maps in Figs.

21j, 21k, and 21l give the impression that removing all lower-level caches somehow clears the interconnect bottleneck, but the poor performance numbers (IPC and memory fetch latency) for these configurations suggest that removing last-level caches shifts the memory bottleneck from the interconnect to DRAM.

5.2.2 Core Isometric Results

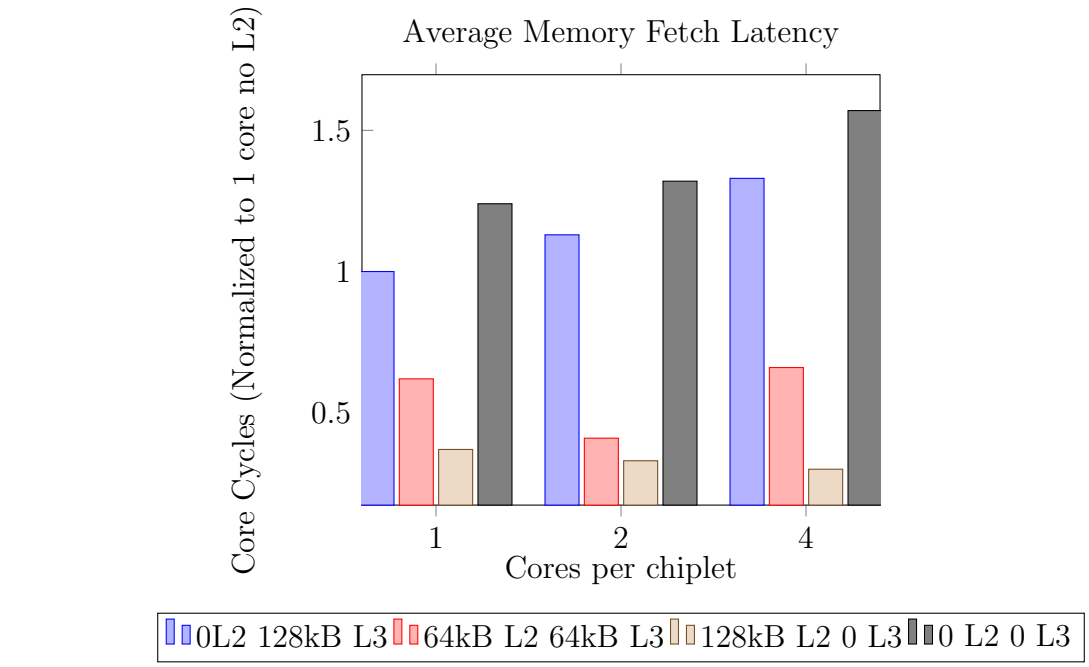
In the core non-isometric experiments, we allowed total core counts to increase with the density of cores per cluster. Because of this, we could keep a 6x6 mesh topology for our interconnect. To isolate the per-core benefits of the chiplet architecture, we must fix the global numbers of memory controllers and SIMT cores (8 and 28, respectively). We must then adapt our interconnect to work around those numbers. Maintaining a square interconnect with 28 SIMT cores, 8 memory controllers, and multiple cores per cluster is not possible without creating mesh nodes with no cores attached. We thus use crossbar interconnects for point-to-point communication between arbitrary numbers of memory controllers and SIMT clusters. This allows for core-isometric evaluations but prevents us from creating interconnect utilization heatmaps. In all other respects, our motivations and processes for collecting core-isometric data are the same as in Section 5.2.1. We find the conclusions from the core-isometric results to be in direct agreement with the conclusions from the core non-isometric results.

Figure 22: Backprop IPC Sensitivity to Core Density and Cache Distribution



In Fig. 22 we see that the 64kB L2 64kB L3, 2 cores per chiplet configuration has roughly 20% higher IPC than the default 0 L2 128kB L3, 1 core per cluster configuration. Furthermore, significant additional L2 cache (as in the 128kB L2 0 L3 configurations) has minimal added benefit.

Figure 23: Backprop Memory Fetch Latency to Core Density and Cache Distribution



As in Fig. 16, our core isometric results show significant decreases in average memory fetch latencies with the addition of L2 cache.

Figure 24: L2 and L3 Cache Miss Rate Sensitivity to Core Density and Cache Distribution

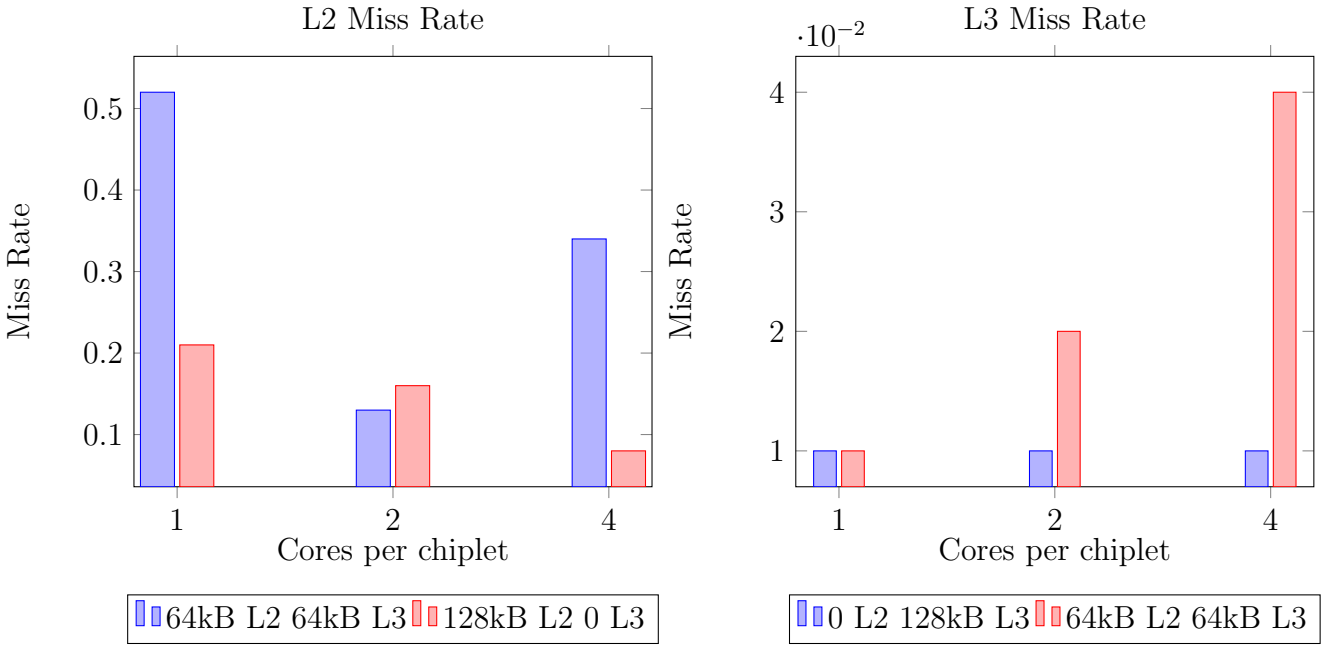


Figure 25: L2 Cache Access and Miss Sensitivity to Core Density and Cache Distribution

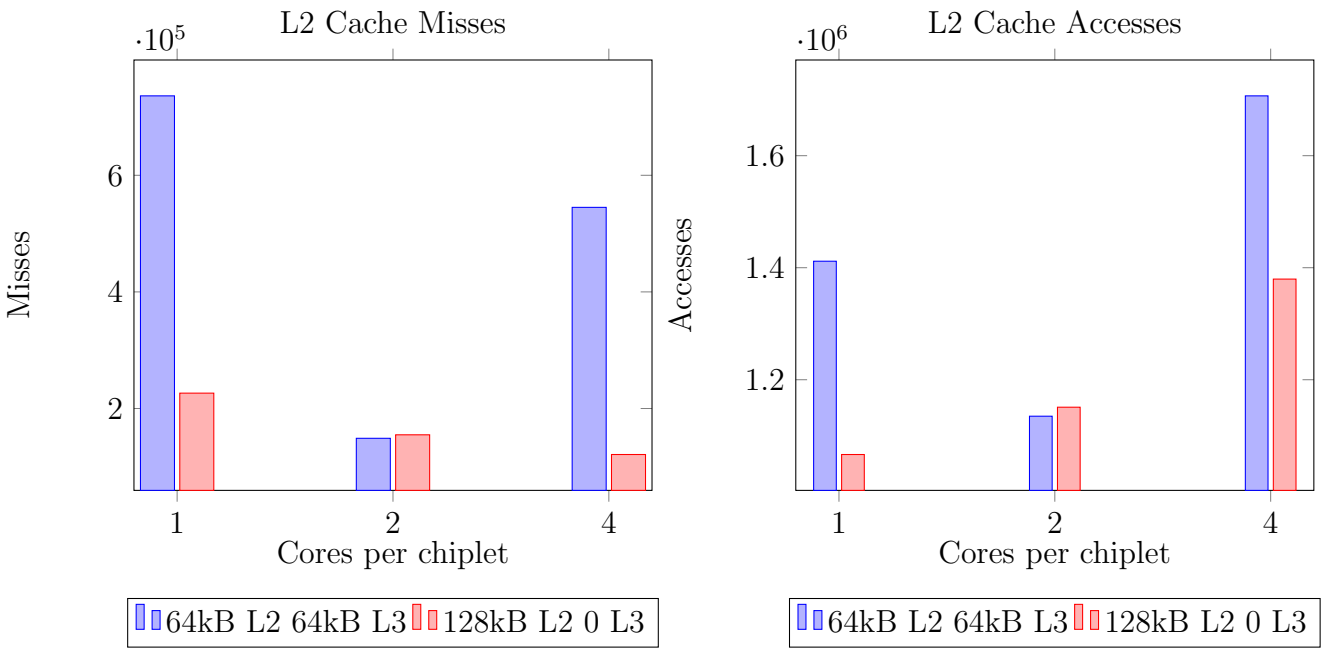


Figure 26: L3 Cache Access and Miss Sensitivity to Core Density and Cache Distribution

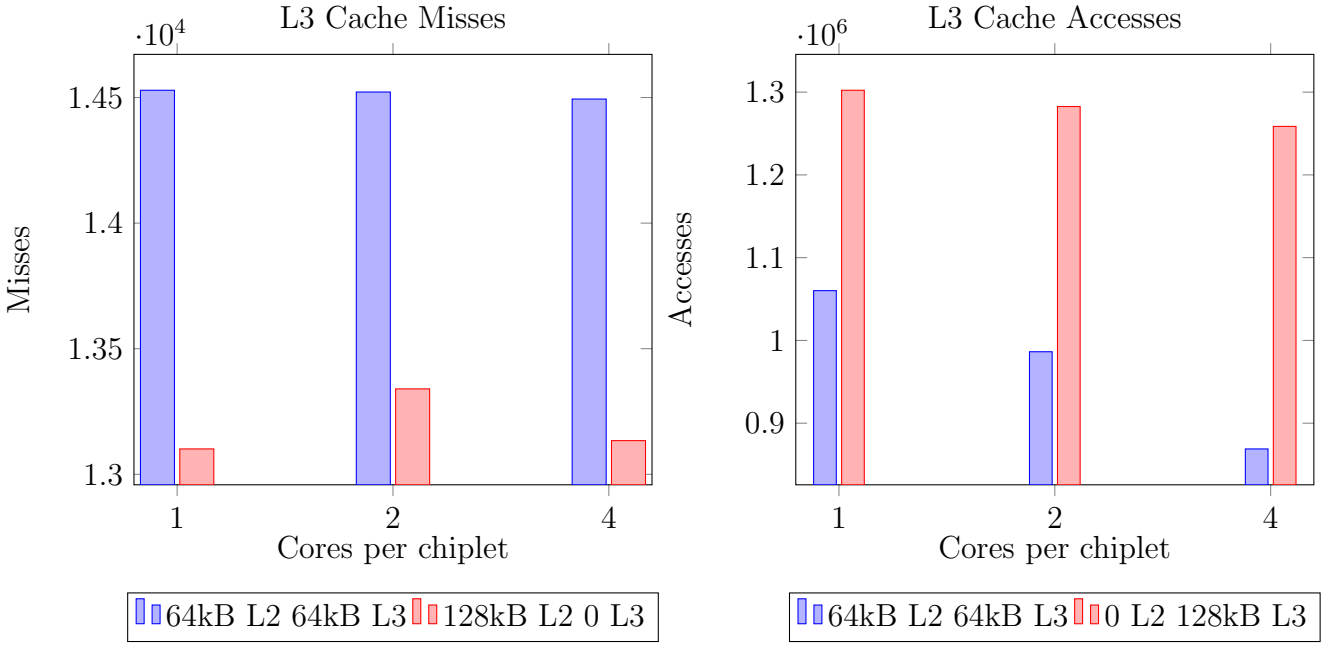
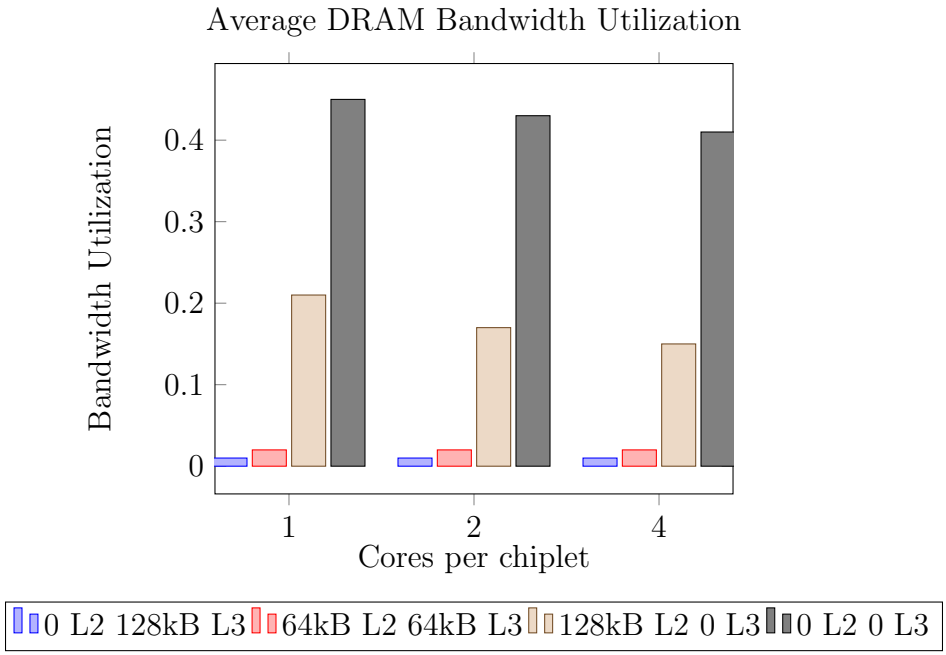


Figure 27: CLW DRAM Utilization Sensitivity to Core Density and Cache Distribution



CLW DRAM utilization is lower in the core-isometric results than in the corresponding core non-isometric results from Fig. 20, but this is expected given the lower total compute power available in the core-isometric configurations. As in the core non-isometric configurations, removing L3 caches caused DRAM bandwidth utilization to spike.

6 Discussion

In implementing chiplets we sought to show that interconnect bottlenecks in neural network backpropagation could be alleviated through the use of an additional layer of cache.

Based on our data, we make the following major observations:

- Neural net backpropagation for fully-connected neural networks can derive significant performance benefits from chiplet architectures with additional levels of on-chip cache (L2). However, the size of the added cache must scale with the number of cores per chiplet.

- Chiplet architectures with sufficient cache can alleviate interconnect hotspots we saw while running backprop. Comparing the reply interconnect utilization without L2 caches to corresponding interconnect utilization with L2 caches, we see that large

increases in IPC can be achieved with relatively small decreases in interconnect utilization.

- There is reason to be concerned that redistributing cache from memory controllers' L3 caches to L2 caches will force GPUs to make heavier use of DRAM during backprop. The MLP used in our experiments are very small compared to the real world networks used for important classification tasks, so it is plausible that training larger MLP without L3 caches would result in DRAM bottlenecks.

- We see no evidence that removing lower-level caches benefits performance for backpropagation on fully-connected neural networks. In our experiments, IPC for trials without lower-level caches was consistently lower than IPC for comparable trials with lower level caches. Without a lower-level cache present, backprop's interconnect bottleneck drained to DRAM and IPC continuously declined with an increase in core per cluster density even as the total core count increased.

From all of this, we conclude that there is use for chiplets in the ML space. However, it must be noted that among ML applications, backpropagation for fully-connected neural networks is far from state of the art. It has become a common pedagogical tool for teaching ML, but precisely because it is an unsophisticated version of more useful neural networks it is not the focus of current study and commercial applications. This is to say that our results cannot broadly justify the use of chiplets in ML. They do, however, establish a jumping-off point for a much wider characterization of other machine learning applications on chiplet architectures.

On the other hand, some neural network variations make use of the fully-connected layers which make up MLP networks. Convolutional neural networks (CNNs) used for commercial image classification, for example, typically use fully-connected layers near the end of the forward-pass. Indeed, operations on these layers often constitute a significant amount of training time for CNNs. So we do see some practical applications for which our results have immediate implications.

7 Related work

Other researchers have had success in alleviating GPU memory bottlenecks. Jia *et.al* demonstrate that GPU caches can be detrimental to the performance of many GPU applications and characterize the impact of L1 caches on the performance of the Rodinia suite benchmarks.

However, they also show that fully-connected neural network backpropagation benefits significantly from added cache [7]. Tian et. al propose techniques for having memory that is unlikely to benefit from temporal locality bypass GPU cache [17]. Singh *et.al* and Wang et. al pursue better cache performance through cache coherence policies [6, 15].

Vijayaranghavan et. al present a vision for Exascale Heterogenous Processors (EHPs) via chiplet-based architectures [20]. Their vision for EHPs balances CPU and GPU chiplets 3D-stacked on an interposer network-on-chip (NOC). To handle the enormous memory demands of such a system, the chiplet architecture is integrated with a network of 3D DRAM technology and external non-volatile memory. Memory overhead is further ameliorated by way of through-silicon via (TSV) connections, electrical connections that pass directly through silicon dies to enable vertical stacking of compute components. The advantages of this chiplet-based vision over previous heterogeneous architectures are described in Section 4.

8 Conclusion

We found that when training MLP neural networks via backpropagation on GPUs a significant memory bottleneck exists at the interconnect network. We simulated a computer architecture based on tiled heterogeneous compute units (chiplets) in an attempt to alleviate this bottleneck. We augmented an existing GPU simulator to achieve a chiplet design and characterize backpropagation under the new architecture. We found that under a chiplet architecture, an appropriately placed and sized second-level cache can have significant impact on the interconnect utilization and application performance of backpropagation for MLPs.

References

[1] "CPU v/s GPU," [Online; accessed 4-10-2017]. [Online]. Available: http://www.e2matrix.com/blog/cpu-vs-gpu/

[2] "Multiprocessor Interconnection Networks," 2013. [Online]. Available: http://15418.courses.cs.cmu.edu/spring2013/article/30

[3] "CUDA," https://en.wikipedia.org/wiki/CUDA, n.d., accessed Mar. 20, 2017.

[4] A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 4 2009.

[5] P. K. Glaskowsky, "NVIDIA Fermi: The First Complete GPU Architecture," http://www.nvidia.com/content/pdf/fermi_white_papers/p.glaskowsky_nvidia's_fermi-the_first_complete_gpu_architecture.pdf, Sept. 2009.

[6] Hao Wang and Vijay Sathish and Ripudaman Singh and Michael Schulte and Nam Sung Kim, "Workload and Power Budget Partitioning for Single-Chip Heterogenous Processors," in *IEEE/ACM Int. Conf. on Parallel Architecture and Compilation Techniques (PACT)*, Sept 2012.

[7] W. Jia, K. A. Shaw, and M. Martonosi, "Characterizing and Improving the Use of Demand-fetched Caches in GPUs," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: ACM, 2012, pp. 15–24. [Online]. Available: http://doi.acm.org/10.1145/2304576.2304582

[8] N. Jiang, J. Balfour, D. U. Becker, B. Towles, W. J. Dally, G. Michelogiannakis, and J. Kim, "A detailed and flexible cycle-accurate network-on-chip simulator," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 86–96.

[9] A. Karpathy, "Hacker's Guide to Neural Networks," http://karpathy.github.io/neuralnets/.

[10] W. Knight, "AI Winter Isn't Coming," Dec. 2016. [Online]. Available: https://www.technologyreview.com/s/603062/ai-winter-isnt-coming/

[11] Y. Lecun, C. Cortes, and C. J. Burges, The MNIST Database, accessed Dec. 1, 2016. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[12] N. Lopes and B. Ribeiro, "An Evaluation of Multiple Feed-Forward Networks on GPUs," *International Journal of Neural Systems (IJNS)*, vol. 21, pp. 31–47, 2011.

[13] D. Reisinger, "Chiplets: The future of circuitry?" April 9 2013.

[14] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "Virtualizing deep neural networks for memory-efficient neural network design," *arXiv preprint arXiv:1602.08124*, 2016.

[15] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt, "Cache coherence for gpu architectures," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2013, pp. 578–590.

[16] A. Tatourian, "NVIDIA GPU Architecture & CUDA Programming Environment," 9 2013. [Online]. Available: https://code.msdn.microsoft.com/windowsdesktop/NVIDIA-GPU-Architecture-45c11e6d

[17] Y. Tian, S. Puthoor, J. L. Greathouse, B. M. Beckmann, and D. A. Jiménez, "Adaptive gpu cache bypassing," in *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, ser. GPGPU-8. New York, NY, USA: ACM, 2015, pp. 25–35. [Online]. Available: http://doi.acm.org/10.1145/2716282.2716283

[18] Tor M. Aamodt and Wilson W.L. Fung and Tayler H. Hetherington, *GPGPU-Sim Manual*, 2009, [Online; accessed 3-20-2017]. [Online]. Available: http://gpgpu-sim.org/manual/images/2/21/Overall-arch.png

[19] P. Tvrdik, "Routing Algorithms and Switching Techniques," http://pages.cs.wisc.edu/~tvrdik/7/html/Section7.html, Dept. of Computer Science, Madison, WI, spring 1999.

[20] T. Vijayaraghavan, Y. Eckert, G. H. Loh, M. J. Schulte, M. Ignatowski, B. M. Beckmann, W. C. Brantley, J. L. Greathouse, W. Huang, A. Karunanithi *et al.*, "Design and Analysis of an APU for Exascale Computing," in *The 23rd IEEE Symposium on High Performance Computer Architecture*. AMD, Dept. of Electrical and Computer Engineering, University of Wisconsin-Madison, Feb. 2017. [Online]. Available: http://www.computermachines.org/joe/publications/pdfs/hpca2017_exascale_apu.pdf